Laravel 9 · База данных · Миграции

- Введение
- Генерация миграций
  - Сжатие миграций
- Структура миграций
- Запуск миграций
  - Откат миграций
- Таблицы
  - Создание таблиц
  - Обновление таблиц
  - Переименование / удаление таблиц
- Столбцы
  - Создание столбцов
  - Доступные типы столбцов
  - Модификаторы столбца
  - Изменение столбцов
  - Удаление столбцов
- Индексы
  - Создание индексов
  - Переименование индексов
  - Удаление индексов
  - Ограничения внешнего ключа
- События

Введение

Миграции похожи на контроль версий для вашей базы данных, позволяют вашей команде определять схемы базы данных приложения и совместно использовать их определение. Если вам когда-либо приходилось указывать товарищу по команде вручную добавить столбец в его схему локальной базы данных после применения изменений в системе управления версиями, то вы столкнулись с проблемой, которую решает миграция базы данных.

Фасад `Schema` обеспечивает независимую от базы данных поддержку для создания и управления таблицами во всех поддерживаемых Laravel системах баз данных. В обычной ситуации, этот фасад используется для создания и изменения таблиц / столбцов базы данных во время миграции.

Генерация миграций

Чтобы сгенерировать новую миграцию базы данных, используйте команду `make:migration` Artisan. Эта команда поместит новый класс миграции в каталог `database/migrations` вашего приложения. Каждое имя файла миграции содержит временную метку, которая позволяет Laravel определять порядок применения миграций:

```
php artisan make:migration create_flights_table
```

Laravel будет использовать имя миграции, чтобы попытаться угадать имя таблицы и будет ли миграция создавать новую таблицу. Если Laravel может определить имя таблицы по имени миграции, то сгенерированный файл миграции будет предварительно заполнен указанной таблицей. В противном случае вы можете просто вручную указать таблицу в файле миграции.

Если вы хотите указать собственный путь для сгенерированной миграции, вы можете использовать параметр `--path` при выполнении команды `make:migration` . Указанный путь должен быть относительно базового пути вашего приложения.

> {tip} Заготовки миграции можно настроить с помощью публикации заготовок.

Сжатие миграций

По мере создания приложения вы можете со временем накапливать все больше и больше миграций. Это может привести к тому, что ваш каталог `database/migrations` станет раздутым из-за потенциально сотен миграций. Если хотите, то можете «сжать» свои миграции в один файл SQL. Для начала выполните команду `schema:dump` :

```
php artisan schema:dump

// Выгрузить текущую схему БД и удалить все существующие миграции ...
php artisan schema:dump --prune
```

Когда вы выполните эту команду, Laravel запишет файл «схемы» в каталог `database/schema` вашего приложения. Теперь, когда вы попытаетесь перенести свою базу данных, Laravel сначала выполнит SQL-операторы файла схемы, при условии, что никакие другие миграции не выполнялись. После выполнения команд файла схемы, Laravel выполнит все оставшиеся миграции, которые не были включены в дамп схемы БД.

Вы должны передать файл схемы базы данных в систему управления версиями, чтобы другие новые разработчики в вашей команде могли быстро воссоздать исходную структуру базы данных вашего приложения.

> {note} Сжатие миграции доступно только для баз данных MySQL, PostgreSQL и SQLite и использует клиент командной строки базы данных. Дампы схемы не могут быть восстановлены в базах данных SQLite, хранимых в памяти.

Структура миграций

Класс миграции содержит два метода: `up` и `down`. Метод `up` используется для добавления новых таблиц, столбцов или индексов в вашу базу данных, тогда как метод `down` должен отменять операции, выполняемые методом `up`.

В обоих этих методах вы можете использовать построитель схем Laravel для выразительного создания и изменения таблиц. Чтобы узнать обо всех методах, доступных построителю `Schema`, просмотрите его документацию. Например, следующая миграция создает таблицу `flights`:

```
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration
{
    /**
     * Запустить миграцию.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('flights', function (Blueprint $table) {
            $table->id();
            $table->string('name');
            $table->string('airline');
            $table->timestamps();
        });
    }

    /**
     * Обратить миграции.
     *
```

```
     * @return void
     */
    public function down()
    {
        Schema::drop('flights');
    }
};
```

Указание соединения миграции

Если ваша миграция будет использовать соединение с базой данных, отличное от соединения с базой данных по умолчанию для вашего приложения, то необходимо установить свойство `$connection` миграции:

```
/**
 * Соединение с БД, которое должно использоваться миграцией.
 *
 * @var string
 */
protected $connection = 'pgsql';

/**
 * Запустить миграцию.
 *
 * @return void
 */
public function up()
{
    //
}
```

Запуск миграций

Чтобы запустить все незавершенные миграции, выполните команду `migrate` Artisan:

```
php artisan migrate
```

Если вы хотите узнать, какие миграции уже выполнены, то вы можете использовать команду `migrate:status` Artisan:

```
php artisan migrate:status
```

Принудительный запуск миграции в рабочем окружении

Некоторые операции миграции являются деструктивными, что означает, что они могут привести к потере данных. Чтобы защитить вас от запуска этих команд для вашей производственной базы данных, от вас потребуется подтверждение перед выполнением команд. Чтобы команды запускались без подтверждения, используйте флаг `--force`:

```
php artisan migrate --force
```

Откат миграций

Чтобы откатить последнюю операцию миграции, вы можете использовать команду `rollback` Artisan. Эта команда откатывает последний «пакет» миграций, который может включать несколько файлов миграции:

```
php artisan migrate:rollback
```

Вы можете откатить ограниченное количество миграций, указав параметр `step` для команды `rollback`. Например, следующая команда откатит последние пять миграций:

```
php artisan migrate:rollback --step=5
```

Команда `migrate:reset` откатит все миграции вашего приложения:

```
php artisan migrate:reset
```

Откат и миграция с помощью одной команды

Команда `migrate:refresh` откатит все ваши миграции, а затем выполнит команду `migrate`. Эта команда эффективно воссоздает всю вашу базу данных:

```
php artisan migrate:refresh

// Обновляем базу данных и запускаем все наполнители базы данных ...
php artisan migrate:refresh --seed
```

Вы можете откатить и повторно запустить ограниченное количество миграций, указав параметр `step` для команды `refresh`. Например, следующая команда откатит и повторно запустит последние пять миграций:

```
php artisan migrate:refresh --step=5
```

Удаление всех таблиц с последующей миграцией

Команда `migrate:fresh` удалит все таблицы из базы данных, а затем выполнит команду `migrate`:

```
php artisan migrate:fresh

php artisan migrate:fresh --seed
```

> {note} Команда `migrate:fresh` удалит все таблицы базы данных независимо от их префикса. Эту команду следует использовать с осторожностью при разработке в базе данных, которая используется совместно с другими приложениями.

Таблицы

Создание таблиц

Чтобы создать новую таблицу базы данных, используйте метод `create` фасада `Schema`. Метод `create` принимает два аргумента: первый – это имя таблицы, а второй – замыкание, которое получает объект `Blueprint`, используемый для определения новой таблицы:

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

Schema::create('users', function (Blueprint $table) {
    $table->id();
    $table->string('name');
    $table->string('email');
    $table->timestamps();
});
```

При создании таблицы вы можете использовать любой из методов столбцов построителя схемы для определения столбцов таблицы.

Проверка наличия таблицы / столбца

Вы можете проверить наличие таблицы или столбца с помощью методов `hasTable` и `hasColumn`, соответственно:

```
if (Schema::hasTable('users')) {
    // Таблица `users` существует ...
}

if (Schema::hasColumn('users', 'email')) {
    // Таблица `users` существует и содержит столбец `email` ...
}
```

Соединение с базой данных и параметры таблицы

Если вы хотите выполнить операцию схемы с подключением, которое не является подключением к базе данных по умолчанию для вашего приложения, используйте метод `connection`:

```
Schema::connection('sqlite')->create('users', function (Blueprint $table) {
    $table->id();
});
```

Кроме того, некоторые другие свойства и методы могут использоваться для определения других аспектов создания таблицы. Свойство `engine` используется для указания механизма хранения таблицы при использовании MySQL:

```
Schema::create('users', function (Blueprint $table) {
    $table->engine = 'InnoDB';

    // ...
});
```

Свойства `charset` и `collation` могут использоваться для указания набора символов и сопоставления для создаваемой таблицы при использовании MySQL:

```
Schema::create('users', function (Blueprint $table) {
    $table->charset = 'utf8mb4';
    $table->collation = 'utf8mb4_unicode_ci';

    // ...
});
```

Метод `temporary` используется, чтобы указать, что таблица должна быть «временной». Временные таблицы видны только текущему сеансу соединения базы данных и автоматически удаляются при закрытии соединения:

```
Schema::create('calculations', function (Blueprint $table) {
    $table->temporary();

    // ...
});
```

Если вы хотите добавить «комментарий» к таблице базы данных, то вы можете вызвать метод `comment` для экземпляра `table` . Комментарии к таблицам в настоящее время поддерживаются только в MySQL и Postgres:

```
Schema::create('calculations', function (Blueprint $table) {
    $table->comment('Business calculations');

    // ...
});
```

Обновление таблиц

Метод `table` фасада `Schema` используется для обновления существующих таблиц. Подобно методу `create`, метод `table` принимает два аргумента: имя таблицы и замыкание, которое получает экземпляр `Blueprint`, используемый для добавления столбцов или индексов в таблицу:

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

Schema::table('users', function (Blueprint $table) {
    $table->integer('votes');
});
```

Переименование / удаление таблиц

Чтобы переименовать существующую таблицу базы данных, используйте метод `rename`:

```
use Illuminate\Support\Facades\Schema;

Schema::rename($from, $to);
```

Чтобы удалить существующую таблицу, вы можете использовать методы `drop` или `dropIfExists`:

```
Schema::drop('users');

Schema::dropIfExists('users');
```

Переименование таблиц с внешними ключами

Перед переименованием таблицы вы должны убедиться, что любые ограничения внешнего ключа в таблице имеют явное имя в ваших файлах миграции, вместо того, чтобы позволять Laravel назначать имя на основе соглашения. В противном случае, имя ограничения внешнего ключа будет ссылаться на имя старой таблицы.

Столбцы

Создание столбцов

Метод `table` фасада `Schema` используется для обновления существующих таблиц. Как и метод `create`, метод `table` принимает два аргумента: имя таблицы и замыкание, которое получает экземпляр `Illuminate\Database\Schema\Blueprint`, используемый для добавления столбцов в таблицу:

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;
```

```
Schema::table('users', function (Blueprint $table) {
    $table->integer('votes');
});
```

Доступные типы столбцов

Построитель схем Blueprint предлагает множество методов, соответствующих различным типам столбцов, которые вы можете добавить в таблицы базы данных. Все доступные методы перечислены в таблице ниже:

- bigIncrements
- bigInteger
- binary
- boolean
- char
- dateTimeTz
- dateTime
- date
- decimal
- double
- enum
- float
- foreignId
- foreignIdFor
- foreignUuid
- geometryCollection
- geometry
- id
- increments
- integer
- ipAddress
- json
- jsonb
- lineString
- longText
- macAddress
- mediumIncrements
- mediumInteger
- mediumText
- morphs

- multiLineString
- multiPoint
- multiPolygon
- nullableMorphs
- nullableTimestamps
- nullableUuidMorphs
- point
- polygon
- rememberToken
- set
- smallIncrements
- smallInteger
- softDeletesTz
- softDeletes
- string
- text
- timeTz
- time
- timestampTz
- timestamp
- timestampsTz
- timestamps
- tinyIncrements
- tinyInteger
- tinyText
- unsignedBigInteger
- unsignedDecimal
- unsignedInteger
- unsignedMediumInteger
- unsignedSmallInteger
- unsignedTinyInteger
- uuidMorphs
- uuid
- year

`bigIncrements()`

Метод `bigIncrements` создает эквивалент автоинкрементного столбца `UNSIGNED BIGINT` (первичный ключ):

```
$table->bigIncrements('id');
```

bigInteger()

Метод `bigInteger` создает эквивалент столбца `BIGINT`:

```
$table->bigInteger('votes');
```

binary()

Метод `binary` создает эквивалент столбца `BLOB`:

```
$table->binary('photo');
```

boolean()

Метод `boolean` создает эквивалент столбца `BOOLEAN`:

```
$table->boolean('confirmed');
```

char()

Метод `char` создает эквивалент столбца `CHAR` указанной длины:

```
$table->char('name', 100);
```

dateTimeTz()

Метод `dateTimeTz` создает эквивалент столбца `DATETIME` (с часовым поясом) с необязательной точностью (общее количество цифр):

```
$table->dateTimeTz('created_at', $precision = 0);
```

dateTime()

Метод `dateTime` создает эквивалент столбца `DATETIME` с необязательной точностью (общее количество цифр):

```
$table->dateTime('created_at', $precision = 0);
```

`date()`

Метод `date` создает эквивалент столбца `DATE`:

```
$table->date('created_at');
```

`decimal()`

Метод `decimal` создает эквивалент столбца `DECIMAL` с точностью (общее количество цифр) и масштабом (десятичные цифры):

```
$table->decimal('amount', $precision = 8, $scale = 2);
```

`double()`

Метод `double` создает эквивалент столбца `DOUBLE` с точностью (общее количество цифр) и масштабом (десятичные цифры):

```
$table->double('amount', 8, 2);
```

`enum()`

Метод `enum` создает эквивалент столбца `ENUM` с указанием допустимых значений:

```
$table->enum('difficulty', ['easy', 'hard']);
```

`float()`

Метод `float` создает эквивалент столбца `FLOAT` с точностью (общее количество цифр) и масштабом (десятичные цифры):

```
$table->float('amount', 8, 2);
```

`foreignId()`

Метод `foreignId` создает эквивалент столбца `UNSIGNED BIGINT`:

```
$table->foreignId('user_id');
```

`foreignIdFor()`

Метод `foreignIdFor` добавляет для переданного класса модели эквивалент столбца `{column}_id UNSIGNED BIGINT` :

```
$table->foreignIdFor(User::class);
```

foreignUuid()

Метод `foreignUuid` создает эквивалент столбца `UUID` :

```
$table->foreignUuid('user_id');
```

geometryCollection()

Метод `geometryCollection` создает эквивалент столбца `GEOMETRYCOLLECTION` :

```
$table->geometryCollection('positions');
```

geometry()

Метод `geometry` создает эквивалент столбца `GEOMETRY` :

```
$table->geometry('positions');
```

id()

Метод `id` является псевдонимом метода `bigIncrements` . По умолчанию метод создает столбец `id` ; однако, вы можете передать имя столбца, если хотите присвоить столбцу другое имя:

```
$table->id();
```

increments()

Метод `increments` создает эквивалент автоинкрементного столбца `UNSIGNED INTEGER` в качестве первичного ключа:

```
$table->increments('id');
```

integer()

Метод `integer` создает эквивалент столбца `INTEGER`:

```
$table->integer('votes');
```

ipAddress()

Метод `ipAddress` создает эквивалент столбца `VARCHAR`:

```
$table->ipAddress('visitor');
```

json()

Метод `json` создает эквивалент столбца `JSON`:

```
$table->json('options');
```

jsonb()

Метод `jsonb` создает эквивалент столбца `JSONB`:

```
$table->jsonb('options');
```

lineString()

Метод `lineString` создает эквивалент столбца `LINESTRING`:

```
$table->lineString('positions');
```

longText()

Метод `longText` создает эквивалент столбца `LONGTEXT`:

```
$table->longText('description');
```

macAddress()

Метод `macAddress` создает столбец, предназначенный для хранения MAC-адреса. Некоторые системы баз данных, такие как PostgreSQL, имеют специальный тип столбца для этого типа данных. Другие системы баз данных будут использовать столбец строкового эквивалента:

```
$table->macAddress('device');
```

mediumIncrements()

Метод `mediumIncrements` создает эквивалент автоинкрементного столбца `UNSIGNED MEDIUMINT` в качестве первичного ключа:

```
$table->mediumIncrements('id');
```

mediumInteger()

Метод `mediumInteger` создает эквивалент столбца `MEDIUMINT`:

```
$table->mediumInteger('votes');
```

mediumText()

Метод `mediumText` создает эквивалент столбца `MEDIUMTEXT`:

```
$table->mediumText('description');
```

morphs()

Метод `morphs` – это удобный метод, который добавляет эквивалент столбца `UNSIGNED BIGINT` (`{column}_id`) и эквивалент столбца `VARCHAR` (`{column}_type`).

Этот метод предназначен для использования при определении столбцов, необходимых для полиморфного отношения Eloquent. В следующем примере будут созданы столбцы `taggable_id` и `taggable_type`:

```
$table->morphs('taggable');
```

multiLineString()

Метод `multiLineString` создает эквивалент столбца `MULTILINESTRING`:

```
$table->multiLineString('positions');
```

multiPoint()

Метод `multiPoint` создает эквивалент столбца `MULTIPOINT`:

```
$table->multiPoint('positions');
```

multiPolygon()

Метод `multiPolygon` создает эквивалент столбца `MULTIPOLYGON`:

```
$table->multiPolygon('positions');
```

nullableTimestamps()

Метод `nullableTimestamps` является псевдонимом для timestamps:

```
$table->nullableTimestamps(0);
```

nullableMorphs()

Метод аналогичен методу `morphs`; тем не менее, создаваемый столбец будет иметь значение NULL:

```
$table->nullableMorphs('taggable');
```

nullableUuidMorphs()

Метод аналогичен методу `uuidMorphs`; тем не менее, создаваемый столбец будет иметь значение NULL:

```
$table->nullableUuidMorphs('taggable');
```

point()

Метод `point` создает эквивалент столбца `POINT`:

```
$table->point('position');
```

polygon()

Метод `polygon` создает эквивалент столбца `POLYGON`:

```
$table->polygon('position');
```

rememberToken()

Метод `rememberToken` создает NULL-эквивалент столбца `VARCHAR(100)`, предназначенный для хранения текущего токена аутентификации:

```
$table->rememberToken();
```

set()

Метод `set` создает эквивалент столбца `SET` с заданным списком допустимых значений:

```
$table->set('flavors', ['strawberry', 'vanilla']);
```

smallIncrements()

Метод `smallIncrements` создает эквивалент автоинкрементного столбца `UNSIGNED SMALLINT` в качестве первичного ключа:

```
$table->smallIncrements('id');
```

smallInteger()

Метод `smallInteger` создает эквивалент столбца `SMALLINT`:

```
$table->smallInteger('votes');
```

softDeletesTz()

Метод `softDeletesTz` добавляет NULL-эквивалент столбца `TIMESTAMP` (с часовым поясом) с необязательной точностью (общее количество цифр). Этот столбец предназначен для хранения временной метки `deleted_at`, необходимой для функции «программного удаления» Eloquent:

```
$table->softDeletesTz($column = 'deleted_at', $precision = 0);
```

softDeletes()

Метод `softDeletes` добавляет NULL-эквивалент столбца `TIMESTAMP` с необязательной точностью (общее количество цифр). Этот столбец предназначен для хранения временной метки `deleted_at`, необходимой для функции «программного удаления» Eloquent:

```
$table->softDeletes($column = 'deleted_at', $precision = 0);
```

string()

Метод `string` создает эквивалент столбца `VARCHAR` указанной длины:

```
$table->string('name', 100);
```

text()

Метод `text` создает эквивалент столбца `TEXT`:

```
$table->text('description');
```

timeTz()

Метод `timeTz` создает эквивалент столбца `TIME` (с часовым поясом) с необязательной точностью (общее количество цифр):

```
$table->timeTz('sunrise', $precision = 0);
```

time()

Метод `time` создает эквивалент столбца `TIME` с необязательной точностью (общее количество цифр):

```
$table->time('sunrise', $precision = 0);
```

timestampTz()

Метод `timestampTz` создает эквивалент столбца `TIMESTAMP` (с часовым поясом) с необязательной точностью (общее количество цифр):

```
$table->timestampTz('added_at', $precision = 0);
```

timestamp()

Метод `timestamp` создает эквивалент столбца `TIMESTAMP` с необязательной точностью (общее количество цифр):

```
$table->timestamp('added_at', $precision = 0);
```

timestampsTz()

Метод `timestampsTz` создает столбцы `created_at` и `updated_at`, эквивалентные `TIMESTAMP` (с часовым поясом) с необязательной точностью (общее количество цифр):

```
$table->timestampsTz($precision = 0);
```

timestamps()

Метод `timestamps` создает столбцы `created_at` и `updated_at`, эквивалентные `TIMESTAMP` с необязательной точностью (общее количество цифр):

```
$table->timestamps($precision = 0);
```

tinyIncrements()

Метод `tinyIncrements` создает эквивалент автоинкрементного столбца `UNSIGNED TINYINT` в качестве первичного ключа:

```
$table->tinyIncrements('id');
```

tinyInteger()

Метод `tinyInteger` создает эквивалент столбца `TINYINT`:

```
$table->tinyInteger('votes');
```

tinyText()

Метод `tinyText` создает эквивалент столбца `TINYTEXT`:

```
$table->tinyText('notes');
```

unsignedBigInteger()

Метод `unsignedBigInteger` создает эквивалент столбца `UNSIGNED BIGINT`:

```
$table->unsignedBigInteger('votes');
```

unsignedDecimal()

Метод `unsignedDecimal` создает эквивалент столбца `UNSIGNED DECIMAL` с необязательной точностью (общее количество цифр) и масштабом (десятичные цифры):

```
$table->unsignedDecimal('amount', $precision = 8, $scale = 2);
```

unsignedInteger()

Метод `unsignedInteger` создает эквивалент столбца `UNSIGNED INTEGER`:

```
$table->unsignedInteger('votes');
```

unsignedMediumInteger()

Метод `unsignedMediumInteger` создает эквивалент столбца `UNSIGNED MEDIUMINT`:

```
$table->unsignedMediumInteger('votes');
```

unsignedSmallInteger()

Метод `unsignedSmallInteger` создает эквивалент столбца `UNSIGNED SMALLINT`:

```
$table->unsignedSmallInteger('votes');
```

unsignedTinyInteger()

Метод `unsignedTinyInteger` создает эквивалент столбца `UNSIGNED TINYINT`:

```
$table->unsignedTinyInteger('votes');
```

uuidMorphs()

Метод `uuidMorphs` – это удобный метод, который добавляет эквивалент столбца `CHAR(36)` (`{column}_id`) и эквивалент столбца `VARCHAR` (`{column}_type`).

Этот метод предназначен для использования при определении столбцов, необходимых для полиморфного отношения Eloquent, использующего идентификаторы UUID. В следующем примере будут созданы столбцы `taggable_id` и `taggable_type`:

```
$table->uuidMorphs('taggable');
```

`uuid()`

Метод `uuid` создает эквивалент столбца `UUID`:

```
$table->uuid('id');
```

`year()`

Метод `year` создает эквивалент столбца `YEAR`:

```
$table->year('birth_year');
```

Модификаторы столбца

В дополнение к типам столбцов, перечисленным выше, есть несколько «модификаторов» столбцов, которые вы можете использовать при добавлении столбца в таблицу базы данных. Например, чтобы сделать столбец «допускающим значение NULL», вы можете использовать метод `nullable`:

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

Schema::table('users', function (Blueprint $table) {
    $table->string('email')->nullable();
});
```

В следующей таблице представлены все доступные модификаторы столбцов. В этот список не входят модификаторы индексов:

Модификатор	Описание
`->after('column')`	Поместить столбец «после» другого столбца (MySQL).
`->autoIncrement()`	Установить столбцы INTEGER как автоинкрементные (первичный ключ).
`->charset('utf8mb4')`	Указать набор символов для столбца (MySQL).

Модификатор	Описание
`->collation('utf8mb4_unicode_ci')`	Указать параметры сравнения для столбца (MySQL/PostgreSQL/SQL Server).
`->comment('my comment')`	Добавить комментарий к столбцу (MySQL/PostgreSQL).
`->default($value)`	Указать значение «по умолчанию» для столбца.
`->first()`	Поместить столбец «первым» в таблице (MySQL).
`->from($integer)`	Установить начальное значение автоинкрементного поля (MySQL / PostgreSQL).
`->invisible()`	Сделать столбец «невидимым» для `SELECT *` -запросов (MySQL).
`->nullable($value = true)`	Позволить (по умолчанию) значения NULL для вставки в столбец.
`->storedAs($expression)`	Создать сохраненный генерируемый столбец (MySQL / PostgreSQL).
`->unsigned()`	Установить столбцы INTEGER как UNSIGNED (MySQL).
`->useCurrent()`	Установить столбцы TIMESTAMP для использования CURRENT_TIMESTAMP в качестве значения по умолчанию.
`->useCurrentOnUpdate()`	Установить столбцы TIMESTAMP для использования CURRENT_TIMESTAMP при обновлении записи.
`->virtualAs($expression)`	Создать виртуальный генерируемый столбец (MySQL).
`->generatedAs($expression)`	Создать столбец идентификаторов с указанными параметрами последовательности (PostgreSQL).
`->always()`	Определить приоритет значений последовательности над вводом для столбца идентификаторов (PostgreSQL).
`->isGeometry()`	Установить тип пространственного столбца как `geometry` , т.е. тип по умолчанию для `geography` (PostgreSQL).

Выражения для значений по умолчанию

Модификатор `default` принимает значение или экземпляр `Illuminate\Database\Query\Expression`. Использование экземпляра `Expression` не позволит Laravel заключить значение в кавычки и позволит вам использовать функции, специфичные для базы данных. Одна из ситуаций, когда это особенно полезно, когда вам нужно назначить значения по умолчанию для столбцов JSON:

```
<?php

use Illuminate\Support\Facades\Schema;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Query\Expression;
use Illuminate\Database\Migrations\Migration;

return new class extends Migration
{
    /**
     * Запустить миграцию.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('flights', function (Blueprint $table) {
            $table->id();
            $table->json('movies')->default(new Expression('(JSON_ARRAY())'));
            $table->timestamps();
        });
    }
};
```

> {note} Поддержка выражений по умолчанию зависит от вашего драйвера базы данных, версии базы данных и типа поля. См. документацию к вашей базе данных. Кроме того, невозможно комбинировать необработанные выражения `default` (используя `DB::raw`) и изменения столбцов через метод `change`.

Порядок столбцов

Метод `after` добавляет набор столбцов после указанного существующего столбца в схеме базы данных MySQL:

```
$table->after('password', function ($table) {
    $table->string('address_line1');
    $table->string('address_line2');
    $table->string('city');
});
```

Изменение столбцов

Необходимые компоненты

Перед изменением столбца вы должны установить пакет `doctrine/dbal` с помощью менеджера пакетов Composer. Библиотека Doctrine DBAL используется для определения текущего состояния столбца и для создания запросов SQL, необходимых для внесения запрошенных изменений в столбец:

```
composer require doctrine/dbal
```

Если вы планируете изменять столбцы, созданные с помощью метода `timestamp`, вы также должны добавить следующую конфигурацию в файл `config/database.php` вашего приложения:

```
use Illuminate\Database\DBAL\TimestampType;

'dbal' => [
    'types' => [
        'timestamp' => TimestampType::class,
    ],
],
```

> {note} Если ваше приложение использует Microsoft SQL Server, то убедитесь, что вы установили `doctrine/dbal:^3.0`.

Обновление атрибутов столбца

Метод `change` позволяет вам изменять тип и атрибуты существующих столбцов. Например, вы можете увеличить размер `string` столбца. Чтобы увидеть метод `change` в действии, давайте увеличим размер столбца `name` до 50. Для этого мы просто определяем новое состояние столбца и затем вызываем метод `change`:

```
Schema::table('users', function (Blueprint $table) {
    $table->string('name', 50)->change();
});
```

Мы также можем изменить столбец, чтобы он допускал значение NULL:

```
Schema::table('users', function (Blueprint $table) {
    $table->string('name', 50)->nullable()->change();
});
```

> {note} Только следующие типы столбцов могут быть изменены: `bigInteger`, `binary`, `boolean`, `char`, `date`, `dateTime`, `dateTimeTz`, `decimal`, `integer`, `json`, `longText`, `mediumText`, `smallInteger`, `string`, `text`, `time`, `unsignedBigInteger`, `unsignedInteger`, `unsignedSmallInteger` и `uuid`.

Переименование столбцов

Чтобы переименовать столбец, вы можете использовать метод `renameColumn` построителя схемы Blueprint. Перед переименованием столбца убедитесь, что вы установили библиотеку `doctrine/dbal` через менеджер пакетов Composer:

```
Schema::table('users', function (Blueprint $table) {
    $table->renameColumn('from', 'to');
});
```

> {note} Переименование `enum` столбца в настоящее время не поддерживается.

Удаление столбцов

Чтобы удалить столбец, вы можете использовать метод `dropColumn` построителя схемы Blueprint. Если ваше приложение использует базу данных SQLite, то вы должны установить библиотеку `doctrine/dbal` через менеджер пакетов Composer, прежде чем использовать метод `dropColumn`:

```
Schema::table('users', function (Blueprint $table) {
    $table->dropColumn('votes');
});
```

Вы можете удалить несколько столбцов из таблицы, передав массив имен столбцов методу `dropColumn`:

```
Schema::table('users', function (Blueprint $table) {
    $table->dropColumn(['votes', 'avatar', 'location']);
});
```

> {note} Удаление или изменение нескольких столбцов в рамках одной миграции при использовании базы данных SQLite не поддерживается.

Доступные псевдонимы команд

Laravel содержит несколько удобных методов, связанных с удалением общих типов столбцов. Каждый из этих методов описан в таблице ниже:

Команда	Описание
`$table->dropMorphs('morphable');`	Удалить столбцы `morphable_id` и `morphable_type`.
`$table->dropRememberToken();`	Удалить столбец `remember_token`.
`$table->dropSoftDeletes();`	Удалить столбец `deleted_at`.
`$table->dropSoftDeletesTz();`	Псевдоним `dropSoftDeletes()`.
`$table->dropTimestamps();`	Удалить столбцы `created_at` и `updated_at`.
`$table->dropTimestampsTz();`	Псевдоним `dropTimestamps()`.

Индексы

Создание индексов

Построитель схем Laravel поддерживает несколько типов индексов. В следующем примере создается новый столбец `email` и указывается, что его значения должны быть уникальными. Чтобы создать индекс, мы можем связать метод `unique` с определением столбца:

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

Schema::table('users', function (Blueprint $table) {
    $table->string('email')->unique();
});
```

В качестве альтернативы вы можете создать индекс после определения столбца. Для этого вы должны вызвать метод `unique` построителя схемы Blueprint. Этот метод принимает имя столбца, который должен получить уникальный индекс:

```
$table->unique('email');
```

Вы даже можете передать массив столбцов методу индекса для создания составного индекса:

```
$table->index(['account_id', 'created_at']);
```

При создании индекса Laravel автоматически сгенерирует имя индекса на основе таблицы, имен столбцов и типа индекса, но вы можете передать второй аргумент методу, чтобы указать имя индекса самостоятельно:

```
$table->unique('email', 'unique_email');
```

Доступные типы индексов

Построитель схем Laravel содержит методы для создания каждого типа индекса, поддерживаемого Laravel. Каждый метод индекса принимает необязательный второй аргумент для указания имени индекса. Если не указано, то имя будет производным от имен таблицы и столбцов, используемых для индекса, а также типа индекса. Все доступные методы индекса описаны в таблице ниже:

Команда	Описание
`$table->primary('id');`	Добавить первичный ключ.

Команда	Описание
`$table->primary(['id', 'parent_id']);`	Добавить составной ключ.
`$table->unique('email');`	Добавить уникальный индекс.
`$table->index('state');`	Добавить простой индекс.
`$table->fullText('body');`	Добавить полнотекстовый индекс (MySQL/PostgreSQL).
`$table->fullText('body')->language('english');`	Добавить полнотекстовый индекс для указанного языка (PostgreSQL).
`$table->spatialIndex('location');`	Добавить пространственный индекс (кроме SQLite).

Длина индекса и MySQL / MariaDB

По умолчанию Laravel использует набор символов `utf8mb4`. Если вы используете версию MySQL древнее 5.7.7 или MariaDB древнее 10.2.2, то вам может потребоваться вручную настроить длину строки по умолчанию, сгенерированную миграциями, чтобы MySQL мог создавать для них индексы. Вы можете настроить длину строки по умолчанию, вызвав метод `Schema::defaultStringLength` в методе `boot` поставщика `App\Providers\AppServiceProvider`:

```
use Illuminate\Support\Facades\Schema;

/**
 * Загрузка любых служб приложения.
 *
 * @return void
 */
public function boot()
{
    Schema::defaultStringLength(191);
}
```

Кроме того, вы можете включить опцию `innodb_large_prefix` для своей базы данных. Обратитесь к документации вашей базы данных для получения инструкций о том, как правильно включить эту опцию.

Переименование индексов

Чтобы переименовать индекс, вы можете использовать метод `renameIndex` построителя схемы Blueprint. Этот метод принимает текущее имя индекса в качестве первого аргумента и желаемое имя в качестве второго аргумента:

```
$table->renameIndex('from', 'to')
```

Удаление индексов

Чтобы удалить индекс, вы должны указать имя индекса. По умолчанию Laravel автоматически назначает имя индекса на основе имени таблицы, имени индексированного столбца и типа индекса. Вот некоторые примеры:

Команда	Описание
`$table->dropPrimary('users_id_primary');`	Удалить первичный ключ из таблицы `users`.
`$table->dropUnique('users_email_unique');`	Удалить уникальный индекс из таблицы `users`.
`$table->dropIndex('geo_state_index');`	Удалить простой индекс из таблицы `geo`.
`$table->dropFullText('posts_body_fulltext');`	Удалить полнотекстовый индекс из таблицы `posts`.
`$table->dropSpatialIndex('geo_location_spatialindex');`	Удалить пространственный индекс из таблицы `geo` (кроме SQLite).

Если вы передадите массив столбцов в метод, удаляющий индексы, то обычное имя индекса будет сгенерировано на основе имени таблицы, столбцов и типа индекса:

```
Schema::table('geo', function (Blueprint $table) {
    $table->dropIndex(['state']); // Удалить простой индекс `geo_state_index`.
});
```

Ограничения внешнего ключа

Laravel также поддерживает создание ограничений внешнего ключа, которые используются для обеспечения ссылочной целостности на уровне базы данных. Например, давайте определим столбец `user_id` в таблице `posts`, который ссылается на столбец `id` в таблице `users`:

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

Schema::table('posts', function (Blueprint $table) {
    $table->unsignedBigInteger('user_id');

    $table->foreign('user_id')->references('id')->on('users');
});
```

Поскольку этот синтаксис довольно подробный, Laravel предлагает дополнительные, более сжатые методы, использующие соглашения, для повышения продуктивности разработки. При использовании метода `foreignId` для создания столбца, пример выше можно переписать так:

```
Schema::table('posts', function (Blueprint $table) {
    $table->foreignId('user_id')->constrained();
});
```

Метод `foreignId` создает эквивалент столбца `UNSIGNED BIGINT`, в то время как метод `constrained` будет использовать соглашения для определения имени таблицы и столбца, на которые ссылаются. Если имя вашей таблицы не соответствует соглашениям Laravel, вы можете указать имя таблицы, передав его в качестве аргумента методу `constrained`:

```
Schema::table('posts', function (Blueprint $table) {
    $table->foreignId('user_id')->constrained('users');
});
```

Вы также можете указать желаемое действие для свойств ограничения «при удалении» и «при обновлении»:

```
$table->foreignId('user_id')
      ->constrained()
      ->onUpdate('cascade')
      ->onDelete('cascade');
```

Для этих действий также предусмотрен альтернативный синтаксис:

Метод	Описание
`$table->cascadeOnUpdate();`	Обновления должны выполняться каскадом.
`$table->restrictOnUpdate();`	Обновления должны быть ограничены.
`$table->cascadeOnDelete();`	Удаление должно происходить каскадом.
`$table->restrictOnDelete();`	Удаление должно быть ограничено.
`$table->nullOnDelete();`	При удалении значение внешнего ключа должно быть установлено как `null`.

Любые дополнительные модификаторы столбца должны быть вызваны перед методом `constrained`:

```
$table->foreignId('user_id')
      ->nullable()
      ->constrained();
```

Удаление внешних ключей

Чтобы удалить внешний ключ, вы можете использовать метод `dropForeign`, передав в качестве аргумента имя ограничения внешнего ключа, которое нужно удалить. Ограничения внешнего ключа используют то же соглашение об именах, что и индексы. Другими словами, имя ограничения внешнего ключа основано на имени таблицы и столбцов в ограничении, за которым следует суффикс `_foreign`:

```
$table->dropForeign('posts_user_id_foreign');
```

В качестве альтернативы вы можете передать массив, содержащий имя столбца, который содержит внешний ключ, методу `dropForeign`. Массив будет преобразован в имя ограничения внешнего ключа с использованием соглашений об именах ограничений Laravel:

```
$table->dropForeign(['user_id']);
```

Переключение ограничений внешнего ключа

Вы можете включить или отключить ограничения внешнего ключа в своих миграциях, используя следующие методы:

```
Schema::enableForeignKeyConstraints();

Schema::disableForeignKeyConstraints();
```

> {note} SQLite по умолчанию отключает ограничения внешнего ключа. При использовании SQLite убедитесь, что включили поддержку внешнего ключа в вашей конфигурации базы данных, прежде чем пытаться создать их в ваших миграциях. Кроме того, SQLite поддерживает внешние ключи только при создании, а не при изменении таблиц.

События

Для удобства каждая операция миграции инициирует событие. Все указанные ниже события расширяют базовый класс `Illuminate\Database\Events\MigrationEvent`:

Класс	Описание
`Illuminate\Database\Events\MigrationsStarted`	Сейчас будет выполнен пакет миграций.

Класс	Описание
`Illuminate\Database\Events\MigrationsEnded`	Выполнение пакета миграций завершено.
`Illuminate\Database\Events\MigrationStarted`	Сейчас будет выполнена одна миграция.
`Illuminate\Database\Events\MigrationEnded`	Выполнение одиночной миграции завершено.
`Illuminate\Database\Events\SchemaDumped`	Дамп схемы базы данных завершен.
`Illuminate\Database\Events\SchemaLoaded`	Загружен существующий дамп схемы базы данных.